NPS52-85-006

NAVAL POSTGRADUATE SCHOOL

Monterey, California

EXPERIENCE WITH Ω:
IMPLEMENTATION OF A
PROTOTYPE PROGRAMMING ENVIRONMENT
PART I

Bruce J. MacLennan

May 1985

Prepared for:

Chief of Naval Research
Arlington, VA 22217

NAVAL POSTGRADUATE SCHOOL
Monterey, California

Rear Admiral R.H. Shumaker
Superintendent

D. A. Schrady
Provost

The work reported herein was supported by Contract N00014-85-WR-24057 from the Office of Naval Research.

This report was prepared by:

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: NPS52-85-006
2. GOVT ACCESSION NO.:
3. RECIPIENT'S CATALOG NUMBER:

4. TITLE (and Subtitle): EXPERIENCE WITH Ω: IMPLEMENTATION OF A PROTOTYPE PROGRAMMING ENVIRONMENT - PART I

5. TYPE OF REPORT & PERIOD COVERED:

6. PERFORMING ORG. REPORT NUMBER:

7. AUTHOR(s): Bruce J. MacLennan

8. CONTRACT OR GRANT NUMBER(s):

9. PERFORMING ORGANIZATION NAME AND ADDRESS: Naval Postgraduate School, Monterey, California 93943

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 61153N; RR014-08-01, N0001485WR24057

11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Arlington, Virginia 22217

12. REPORT DATE: May 1985

13. NUMBER OF PAGES:

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office):

15. SECURITY CLASS. (of this report): unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE:

16. DISTRIBUTION STATEMENT (of this Report):

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report): Approved for public release, distribution uhlimited

18. SUPPLEMENTARY NOTES:

19. KEY WORDS (Continue on reverse side if necessary and identify by block number): Object-oriented programming, production rules, production systems, entity-relationship, pseudo-natural language, knowledge representation, natural language interface, logic programming, simulation language, rule-based system, knowledge base, programming environment, software prototyping, rapid prototyping, syntax-directed editor, unparser

20. ABSTRACT (Continue on reverse side if necessary and identify by block number):

This is the first report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report describes the interpreter, unparser, syntax directed editor, command interpreter and debugger for a subset of the language, namely arithmetic expressions.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S N 0102-LF-014-6601

EXPERIENCE WITH Ω

IMPLEMENTATION OF A

PROTOTYPE PROGRAMMING ENVIRONMENT

PART I

Bruce J. MacLennan
Computer Science Department
Naval Postgraduate School
Monterey, CA 93943

Abstract:

This is the first report of a series exploring the use of the Ω programming notation to prototype a programming environment. This environment includes an interpreter, unparser, syntax directed editor, command interpreter, debugger and code generator, and supports programming in a small applicative language. The present report describes the interpreter, unparser, syntax directed editor, command interpreter and debugger for a subset of the language, namely arithmetic expressions.

1. Introduction

Our goal is to explore in the context of a very simple language the use of the Ω programming notation [MacLennan83, MacLennan85] to implement some of the tools that constitute a programming environment. Succeeding reports will extend the tools described in this report*.

The structure of this report is as follows: First we briefly review the Ω programming notation for describing transformations on relations. Second, we define a simple language — the language of arithmetic expressions. An abstract syntax for this language is defined in terms of relations. Third, we discuss abstract interpretation of programs in this language. Fourth, we modify the interpreter to accomplish unparsing. Next, we look at error recovery and interactive debugging. Finally, we consider syntax directed editing of abstract programs.

* Support for this research was provided by the Office of Naval Research under contract N00014-85-WR-24057.

2. OVERVIEW OF Ω

2.1 Requirements for a Programming Environment Database

To understand the relevance of Ω to programming environments we begin by stating the requirements for a software development database. It will be required to store and interrelate many kinds of information:

- Programs
- Specifications
- Documentation
- Version Information
- Comments
- Object Code
- Implementation Hints
- Test Data
- Test Results
- Reports
- Runtime Structures

In prototyping tools and environments, we want to make a minimum of implementation commitments. Hence, it is convenient to take a *relational* view. This is because relations are a well-understood, implementation-independent way of viewing databases.

2.2 RELATIONS

We will view the computer system (or network) as containing a (possibly large) number of (finite) relations. The number of relations is not fixed; new (empty) relations can be created by direct (user) or indirect (program) request. The tuples in relations can contain:

- Values: For example, numbers, Booleans, characters, strings, "pure" lists.
- Objects: Essentially unique IDs; their only properties are the relations in which they participate.

Relations are themselves objects.

2.3 Notation

We use the notation '$R(x,y,\ldots,z)$' to mean the tuple $(x,y,\ldots,z)$ is in R. Similarly, '$\neg R(x,y,\ldots,z)$' means that there is no such tuple in R.

Operations on the database are described by a kind of production rule:

$$cause \implies effect$$

The effect part is composed of a series of *transactions*. There are two forms for a transaction:

- $R(x,y,\ldots,z)$ means add the tuple $(x,y,\ldots,z)$ to R.
- $\neg R(x,y,\ldots,z)$ means delete the tuple from R.

The arguments x, y, ..., z can be any applicative expressions. They can also be *procedures*, i.e., parameterized database transformations.

The cause part has the form:

$$condition\ ,\ \cdots$$

The cause is a sequence of conditions separated by commas. Each condition asks whether certain relations hold certain tuples.

The condition '$R(x,y,\ldots,z)$' *succeeds* if there is a tuple $(x,y,\ldots,z)$ in R. On the other hand, '$\neg R(x,y,\ldots,z)$' *fails* if there is such a tuple in R. The arguments x, y, ..., z have the forms:

- a *constant* matches itself;
- a *free variable* matches anything, and becomes bound to value it matches.
- an *applicative expression* matches its value.

We consider several examples of conditions. $R(2,3)$ succeeds if the pair $(2,3)$ *is* in R. $\neg R(2,3)$ succeeds if the pair $(2,3)$ *isn't* in R. If 'y' is free (unbound), then $R(2,y)$ succeeds if there is a pair of

the form $(2,v)$ in R; variable 'y' becomes bound to v. $\neg R(2,y)$ succeeds if there is *no* pair of the form $(2,v)$ in R. If 'y' is bound to v, then $S(y,3)$ succeeds if the pair $(v,3)$ is in S. $\neg S(y,3)$ succeeds if the pair $(v,3)$ is *not* in S.

An implicit join is a rule in which the same free variable appears in several conditions. For example,

$$R(2,y),\ S(y,3)$$

succeeds if:

1. There is a pair of the form $(2,v)$ in R
2. The pair $(v,3)$ is in S

It fails, otherwise. Note: There may be many pairs of the form $(2,v)$ in R. The join succeeds if for one or more of them $(v,3)$ is in S. Any conjunction of conditions is allowed in the cause.

2.4 Stack Example

To illustrate these ideas we show the Ω definition of a stack manager. First we introduce the relations and their intuitive meanings:

- Stack(s) – object s is a stack
- Contents(x,s) – the list x is the contents of s
- Push(a,x,s) – a pushes x on s
- Pop(a,s) – a pops s
- Receives(a,x) – a receives x

The domains of these relations are described by the following assertions (second order relations). Thus 'Degree (Contents, 2)' means that all the tuples in Contents have two elements; 'Domain (list, 1, Contents)' means that the first elements of the tuples in Contents satisfy the 'list' predicate; and 'Indexed (Contents, 2)' means that the second elements of the tuples in Contents are all unique. Each assertion is shown in two notations: the usual predicate notation and a pseudonatural notation (Ω supports several semantically equivalent notations; see [MacLennan84, Ufford85]).

- Degree (Stack, 1)

  'Stack' has degree 1.

- Degree (Contents, 2),

  'Contents' has degree 2.

- Domain (list, 1, Contents)

  'list' is domain 1 of 'Contents'.

- Domain (Stack, 2, Contents)

  'Stack' is domain 2 of 'Contents'.

- Indexed (Contents, 2)

  'Contents' is indexed by domain 2.

- Degree (Push, 3)

  'Push' has degree 3.

- Domain (Stack, 3, Push)

  'Stack' is domain 3 of 'Push'.

- Degree (Pop, 2)

  'Pop' has degree 2.

- Domain (Stack, 2, Pop)

  'Stack' is domain 2 of 'Pop'.

- Degree (Receives, 2)

  'Receives' has degree 2.

The pop rule describes how to pop a stack:

Stack (s), Pop (a,s), Contents (x,s)

$\Rightarrow$ $\neg$Pop (a,s),

$\neg$Contents (x,s),

Receive (a, first [x]),

Contents (rest [x], s)

It can be read as follows: "If s is a stack, a is popping s, and x is the contents of s, then a is not popping s, x is not the contents of s, a receives the first element of x, and the rest of x is the contents of s."

The push rule is analogous:

Stack (s), Push (a,x,s), Contents (y,s)

$\Rightarrow$ ¬Push (a,x,s),

¬Contents (y,s),

Receive (a,s),

Contents (cons [x,y], s)

If s is a stack, a is pushing x on s, and y is the contents of s, then a is not pushing x on s, y is not the contents of s, a receives s, and the result of consing x on y is the contents of s.

2.5 Further Notational Conventions

Notice that in the push and pop rules, conditions found to hold in the cause parts of rules are often canceled in the effect parts. Since this is a very common situation we introduce the following *cancellation convention:* When a tuple found in the condition is removed by the effect, i.e., a relation that holds in the condition is canceled by the effect, we can indicate this by an '*' before the condition. For example, the pop and push rules can be written:

Stack(s), *Pop(a,s), *Contents(x,s)

$\Rightarrow$ Receive(a, first[x]), Contents(rest[x], s)

Stack(s), *Push(a,x,s), *Contents(y,s)

$\Rightarrow$ Receive(a,s), Contents(cons[x,y], s)

It is often useful to limit the application of rules by *constraints,* which are implemented as follows. The relation called '**if**' contains the single value **true**. Hence, an applicative expression that evaluates to a Boolean value can be used to constrain rule application. For example:

$$\text{Sched}\ (x.t),\ \text{Clock}\ (t'),\ \textbf{if}\ (t' \geqslant t) \implies \cdots$$

For convenience we often omit the **if** and write:

$$\text{Sched}\ (x,t),\ \text{Clock}\ (t'),\ t' \geqslant t \implies \cdots$$

We can summarize all that we've seen in the following (simplified) grammar for the Ω language:

Simplified Ω Grammar

$$\begin{aligned}
\mathit{rule} &= [\,\mathit{cause} \implies]\ \mathit{effect} \\
\mathit{cause} &= \mathit{cond}\ ,\ \cdots \\
\mathit{cond} &= \begin{bmatrix} * \\ \neg \end{bmatrix}\ \mathit{rel}\ \mathit{args} \\
\mathit{args} &= (\ \mathit{expr}\ ,\ \cdots\ ) \\
\mathit{effect} &= \mathit{trans}\ ,\ \cdots \\
\mathit{trans} &= [\neg]\ \mathit{rel}\ \mathit{args}
\end{aligned}$$

A complete Ω grammar can be found in [MacLennan85].

2.6 Pseudonatural Notation

We have experimented with several pseudonatural notations for Ω rules. The notation used in this report is a variation of that described in [MacLennan84] and [Ufford85]. Relations can be named by templates, for example:

- is stack
- is contents of –
- pops –
- pushes – on –
- receives –

Rules are written:

If – then –

with ',' or ', and' for ','. The word 'given' represents the cancellation convention '*'. Hence the pop

rule can be written:

If S is stack, given A pops S, and given X is contents of S

then A receives first of X. and rest of X is contents of S.

The push rule is:

If S is stack, given A pushes X on S, and given Y is contents of S

then A receives S and catenation of X and Y is contents of S.

Allowing 'a' 'an' and 'the' as noise words and using words for variable names we have:

If an object is a stack, given an agent pops the object, and given a list is the contents of the object then the agent receives the first of the list, and the rest of the list is the contents of the object.

If an object is a stack, given an agent pushes a thing on the stack, and given a list is the contents of the stack then the agent receives the stack, and the catenation of the thing and the list is the contents of the stack.

3. Abstract Structure

For our example programming environment we will use a very simple language of arithmetic expressions composed of $+$, $-$, $\times$, $\div$, parentheses and literal integers. A typical program is:

$$(3+5) \times 6$$

First we must define an abstract structure for representing programs. There are two kinds of nodes:

- Constant Nodes: correspond to literals.
- Application Nodes: correspond to the application of an operator to its operands

These nodes and their interconnections can be represented by the relations:

- Con (E)

  E is a constant

- Litval (V, E)

  V is the literal_value of E.

- Appl (E)

  E is an application.

- Op (F, E)

  F is the operator of E.

- Left (X, E)

  X is the left_argument of E.

- Right (Y, E)

  Y is the right_argument of E.

For example, the program

$$(3+5) \times 6$$

would be represented by the database:

Appl (n1)

Op ("×", n1)

Left (n2, n1)

Right (n3, n1)

Appl (n2)

Op ("+", n2)

Left (n4, n2)

Right (n5, n2)

Con (n3)

Litval (6, n3)

Con (n4)

Litval (3, n4)

Con (n5)

Litval (5, n5)

Meaning (sum, "+")

Meaning (product, "×")

We have given the objects names ('n1', 'n2', etc.) only so they can be referred to in our example; normally they would be anonymous since the tree would have been constructed by an editor. This database is portrayed in Figure 1.

In defining the domains of the various relations, it will be convenient to make use of the following function abbreviation. Let 'Function(F,D,R)' mean

- Degree(F,2),
- Domain(R,1,F),
- Domain(D,2,F),
- Indexed(F,2).

In the pseudonatural notation we can say:

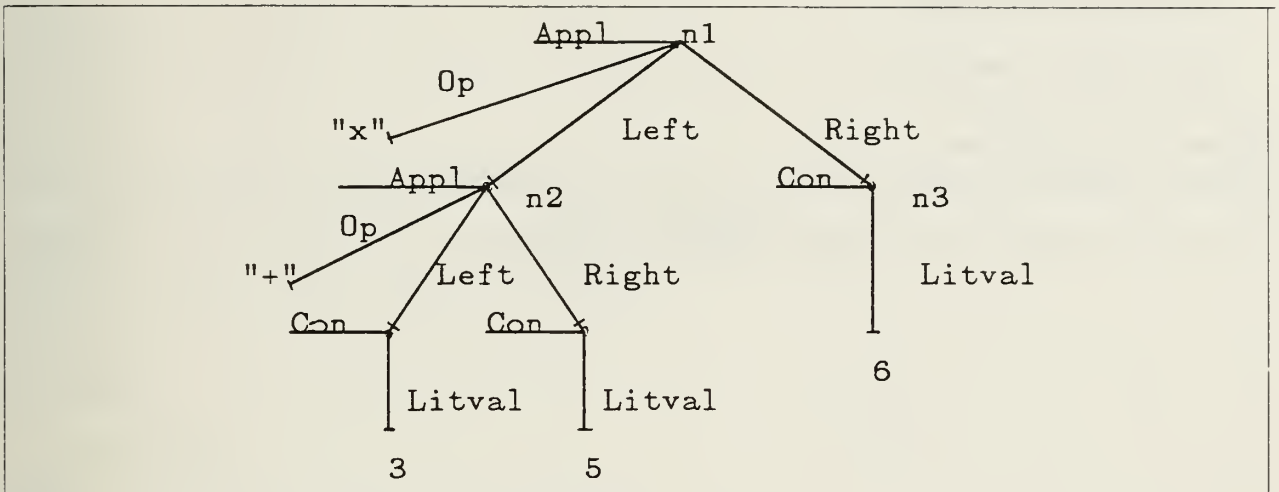


Figure 1. Database Representing Abstract Program

'F is a function from D to R' means

'F has degree 1, R is domain 1 of F, D is domain 2 of F, and F is indexed on domain 2'.

Hence, if we know that Function(F,D,R), that is, that F is a function from D to R, then we know that given any x in D there will be at most one y such that $F(y,x)$.

The domains of the abstract program structure relations (in the pseudonatural notation) are as follows:

- 'constant' has degree 1.
- 'literal value' is a function from 'constant' to 'integer'.
- 'application' has degree 1.
- 'operator' is a function from 'application' to 'string'.
- 'left_argument' is a function from 'application' to 'expression'.
- 'right_argument' is a function from 'application' to 'expression'.

In the predicate notation this is:

Degree (Con, 1)

Function (litval, Con, integer)

Degree (Appl, 1)

Function (Op, Appl, string)

Function (Left, expr, expr)

Function (Right, expr, expr)

Here we have assumed that

$$expr = Con \cup Appl$$

4. Evaluation

4.1 Relations

The preceding relations define the abstract program structure; they are *static* (with respect to program evaluation). We need additional relations to control program evaluation; they are *dynamic* (with respect to program evaluation). The evaluation relations are:

- Eval (E)

  E is evaluated

- Value (V, E)

  V is the value of E

- Meaning (F, N)

  F is the meaning of N.

The domains of these relations are:

- Degree(Eval,1), Domain(expr,1,Eval).

  'evaluated' has degree 1, and 'expression' is domain 1 of 'evaluated'.

- Function (Value, expr, integer)

  'value' is a function from 'expression' to 'integer'.

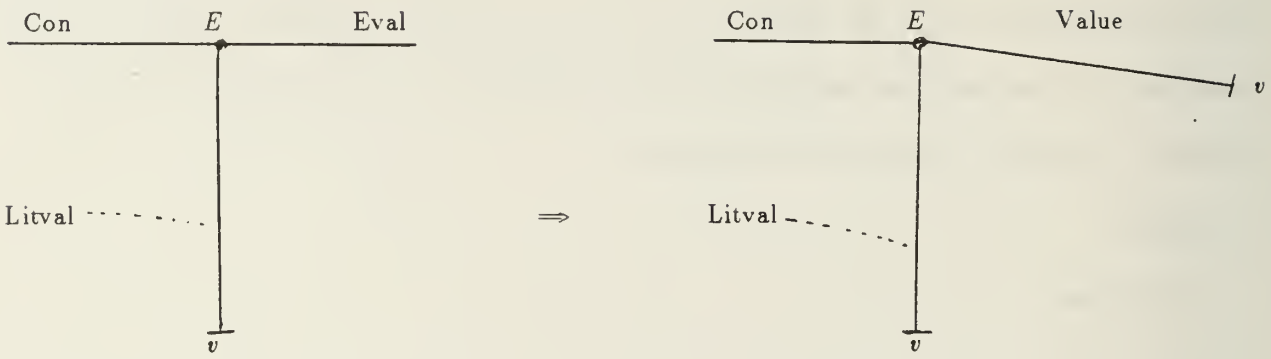
  Function (Meaning, string, function)

  'meaning' is a function from 'string' to 'expression'.

Eval and Value can be thought of as *attributes* that at various times are attached to various nodes in the tree. In particular, the Eval attribute on a node means that the evaluation of that node has been requested, but not serviced. The Value attribute associates a value with a node until such time as that value is used. The Meaning relation is a table that maps the names of operators into functions for performing the operations.

4.2 Evaluation of Constant Node

The transformation required to evaluate a constant node can be portrayed:



That is, if the Eval attribute arrives at a constant node, then we remove the Eval attribute (since the request has been serviced), and use a Value link to bind the node's literal value to the node. In other words, when an Eval arrives at a leaf of the tree it is converted into a Value, which will travel back up the tree.

The rule for accomplishing this is simply[1]:

If given an expression is evaluated, the expression is a constant, a number is the literal value of the expression, and a function is the meaning of "lit"

then the function of the number is the value of the expression.

This is expressed in the predicate notation as:

$$\text{*Eval}(E),\ \text{Con}(E),\ \text{Litval}(V,E),\ \text{Meaning}(F,\text{"lit"}) \implies \text{Value}(F[V],E)$$

The function that is the meaning of "lit" is the identity function, hence the following simpler rule would also work:

$$\text{*Eval}(E),\ \text{Con}(E),\ \text{Litval}(V,E) \implies \text{Value}(V,E)$$

We have used the more complicated rule to display the symmetry between the evaluator and the unparser (which is discussed in the following section).

1. The pseudonatural transcription of the rules was prepared by Robert Ufford; see [Ufford85].

4.3 Evaluation of Application Node

Evaluation of application nodes is accomplished in two steps. First there is a downward *analysis* pass when the Eval attribute reaches the node and is passed to its daughters. Later there is an upward *synthesis* pass when the Values from the daughters arrive back at the node and are used to compute the Value for the node. These steps can be visualized as in Figures 2 and 3.

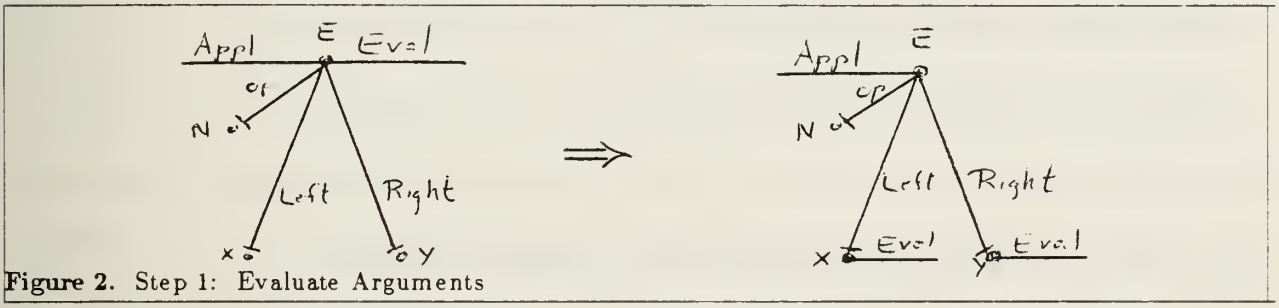


Figure 2. Step 1: Evaluate Arguments

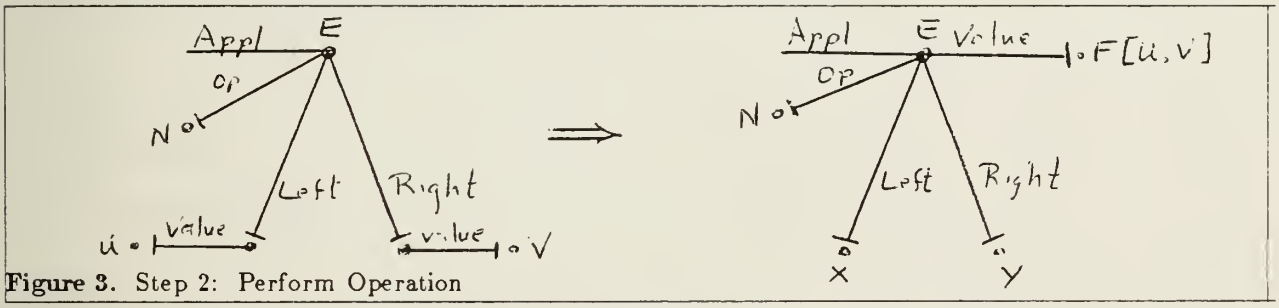


Figure 3. Step 2: Perform Operation

It is easy to translate these diagrams into rules. First the analysis (downward) rule:

> If given an expression is evaluated, the expression is an application, node_1 is the left_argument of the expression, and node_2 is the right_argument of the expression
>
> then node_1 is evaluated, and node_2 is evaluated.

In the predicate notation:

> *Eval(E), Appl(E), Left(X,E), Right(Y,E)
>
> $\Rightarrow$ Eval(X), Eval(Y)

Notice that the Eval flag is passed to both daughters simultaneously; thus they can be evaluated concurrently or in any other order. Hence, Eval is a *parallel evaluator*.

Next we consider the synthesis (upward) rule:

If an expression is an application, a string is the operator of the expression, node_1 is the left argument of the expression, node_2 is the right argument of the expression, a function is the meaning of the string, given number_1 is the value of node_1 and given number_2 is the value of node_2 then the function of number_1 and number_2 is the value of the expression.

In the predicate notation:

$$\text{Appl}(E),\ \text{Op}(N,E),\ \text{Left}(X,E),\ \text{Right}(Y,E),$$

$$\text{Meaning}(F,N),\ {}^{*}\text{Value}(U,X),\ {}^{*}\text{Value}(V,Y)$$

$$\Rightarrow\ \text{Value}\ (F\ [U,V],\ E)$$

This completes the parallel evaluator for simple abstract arithmetic expressions.

5. Unparsing

5.1 Relations

We will now use exactly the same approach as used for evaluation, but for a different purpose – unparsing. Instead of computing an *integer value* for the abstract program, we will compute a *string value,* which is the program's concrete representation. That is we will *unparse* the abstract program. This is accomplished by changing the interpretation of the literal constants and the primitive operators.

This is a quite general approach. From a single program such as the evaluator we can generate a *family* of related programs, just by changing the domain of interpretation of the constants and operators. Examples of tools amenable to this approach are unparsers, type checkers, symbolic evaluators.

The following relations are needed:

- Unparse (E)

  E is unparsed

  (corresponds to Eval)

- Image (S, E)

  S is the image of E

  (corresponds to Value)

- Template (T, N)

  T is the template for N

  (corresponds to Meaning)

Given our previous example, Unparse(n1) will eventually result in

Image ("((3+5)×6)", n1)

For simplicity we have generated an image that is fully parenthesized.

The domains of the relations are as follows:

- Degree (Unparse, 1)

  'unparsed' has degree 1.

- Function (Image, expr, string).

  'image' is a function from 'expression' to 'string'.

- Function (Template, string, function).

  'template' is a function from 'string' to 'function'.

We also will make the following assumptions:

- Assume the function 'string← int [n]' converts the integer n into a string.
- Assume Template (string← int, "lit").
- Assume $s \hat{} t$ represents string catenation.
- For operator symbols N, assume Template (F, N), where

$$F[U, V] = \text{"("} \hat{} U \hat{} N \hat{} V \hat{} \text{")"}$$

  and N is "+" or "-" or "×" or "÷". Thus, if Template (F, "+"), then

$$F[\text{"3"}, \text{"(6} \times \text{2)"}] = \text{"(3 + (6} \times \text{2)"}$$

5.2 Unparsing Constants

When the Unparse attribute arrives at a constant node, the literal value is converted to a string and made the image of the node. In the pseudonatural notation the rule is:

If given an expression is unparsed, the expression is a constant, a number is the literal value of the expression, and a function is the template of "lit"

then the function of the number is the image of the expression.

In predicate notation it is:

*Unparse(E), Con(E), Litval(V,E), Template(F,E)

$\Rightarrow$ Image($F[V]$, E)

5.3 Unparsing Applications

The arrival of the unparse attribute at an application node triggers its propagation to the daughter nodes. Hence the analysis rule in pseudonatural notation is:

If given an expression is unparsed, the expression is an application, node-1 is the left_argument of the expression, and node_2 is the right_argument of the expression

then node_1 is unparsed, and node_2 is unparsed.

In predicate notation it is:

*Unparse(E), Appl(E), Left(X,E), Right(Y,E),

$\Rightarrow$ Unparse(X), Unparse(Y)

When images arrive at the daughters of the application, they are combined by the template function of the operator into an image for the entire node. Hence the synthesis rule in pseudonatural notation is:

If an expression is an application, a string is the operator of the expression, node_1 is the left_argument of the expression, node_2 is the right_argument of the expression, a function is the template of the string, given image_1 is the image of node_1, and given image_2 is the image of node_2

then the function of image_1 and image_2 is the image of the expression.

In the predicate notation:

Appl(E), Op(N,E), Left(X,E), Right(Y,E),

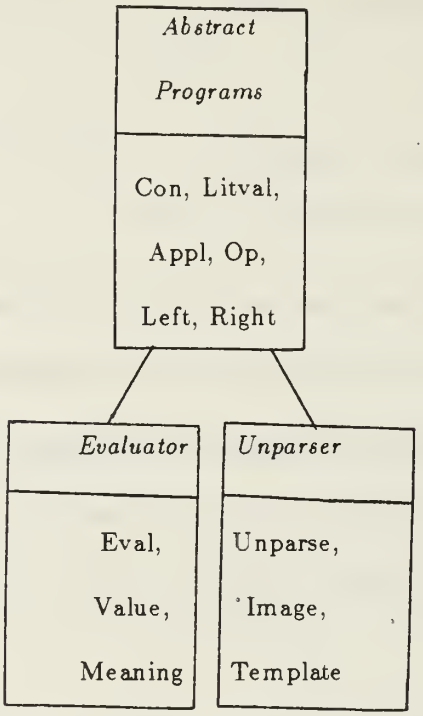
Template(F,N), *Image(U,X), *Image(V,Y)

$\Rightarrow$ Image (F [U,V], E)

This completes the rules for the unparser.

6. Information Hiding

The relations can be divided into three domains of accessibility on the basis of "need to know":



That is, both the evaluator and the unparser need access to the abstract structure relations (read-only access, actually). On the other hand, the evaluator needs access to its own dynamic relations, but not to those of the unparser. Conversely, the unparser needs access to its own relations, but not the evaluator's. These access restrictions can be enforced by the capability mechanism described in [MacLennan85].

7. Error Handling

7.1 Error Detection

A real programming environment must be able to detect errors and allow the user to deal with them. Let's consider what happens when an error occurs. Suppose we evaluate $((3 \div 0)+1)$. This leads to the assertion

$$\text{Value (quotient } [3,0], n)$$

Suppose quotient$[3,0] \Longrightarrow$ error. This will trigger that assertion

$$\text{Value (sum } [\text{error},1], m)$$

If we further assume sum$[\text{error},1] \Longrightarrow$ error, then the value returned for the entire evaluation is 'error'. This result is not helpful – it doesn't tell us *where* the error occurred, only *that* an error occurred.

One possible solution is to suppose the primitive operations return "error codes" when something goes wrong, that the code indicates what went wrong, and that error codes are distinguishable from integers and other "legitimate" values. Then we can incorporate error checking into our interpreter. We do this by tentatively attaching the alleged value to the node (via a new relation called Check) until it is determined whether or not the value is legitimate. This requires the following additional relations:

- Check (V, E)

  V is the value to be checked for E

- Explanation (S, C)

  S is the explanation of error C

- Display (S)

  S is displayed

- CurrentNode (E)

  E is the current node

The domains of these relations are:

Function (Check. expr, integer),

Function (Explanation, errorcode. string),

Degree (Display, 1).

Domain (string, 1, Display).

We then need to modify the second (upward or synthesis) rule for applications. It is replaced by these three rules:

Appl(E), Op(N,E), Left(X,E), Right (Y,E),

Meaning(F,N), *Value(U,X), *Value(V, Y)

$\Rightarrow$ Check (F [U, V], E).

*Check(W,E), **if** (integer[W]) $\Rightarrow$ Value(W,E).

*Check(W,E), Explanation(S,W),

*CurrentNode(–), **if** (errorcode[W])

$\Rightarrow$ Display{S}, CurrentNode(E).

The first rule tentatively attaches $F[U,V]$ to the application node via Check. If the value of $F[U,V]$ is an integer, then the second rule converts the Check connection to a Value connection to reflect the fact that the value is bona fide. On the other hand, if the value of $F[U,V]$ is an error code, then evaluation is stopped (since no Value is attached to the node), the offending node is recorded (in CurrentNode), and an explanation of the error code is displayed.

In the pseudonatural notation these rules are:

If an expression is an application, a string is the operator of the expression, node_1 is the left_argument of the expression, node_2 is the right_argument of the expression, a function is the meaning of the string, given number_1 is the value of node_1, and given number_2 is the value of node_2 then the function of number_1 and number_2 is the value to be checked for the expression.

If given an alleged_value is the value to be checked for an expression, and the alleged_value is an integer then the alleged_value is the value of the expression.

If given an alleged_value is the value to be checked for an expression, the alleged_value is an error_code, a string is the explanation of the error_code, and any_node is the current_node then the string is displayed. and the expression is the current_node.

Thus, if an error value is detected, evaluation is suspended, an error message is issued, and the offending node is recorded.

The alert reader will realize that with a parallel evaluator there is the possibility of several errors occuring concurrently. With the error checking method presented above, all the diagnostics will be issued correctly, but CurrentNode will record only the *last* node to generate an error. A more elaborate system would place all error nodes in a two place relation ErrorNodes such that

$$\text{ErrorNodes}\ (E, C)$$

means that node E generated error code C. It is then necessary to have a command for removing a node from ErrorNodes and making it the CurrentNode. The detailed implementation is left as an exercise for the reader.

7.2 Suspension

What is the state when an error message is sent? There may be parallel computations in progress, but since no Value has been provided for E, the evaluation cannot complete. It is *suspended,* waiting for a value for E.

There are several possible actions:

1. Supply a value for the offending node and let evaluation continue. E.g.[2],

$$\text{CurrentNode}\ (E) \implies \text{Value}\ (0, E);$$

2. Unparse the offending node to find the problem:

$$\text{CurrentNode}\ (E) \implies \text{Unparse}\ (E);$$

$$\text{CurrentNode}\ (E),\ \text{*Image}\ (S, E) \implies \text{Display}\ \{S\};$$

2. We show the rule that would be typed in Ω *command mode* to effect the desired error recovery action.

3. Investigate neighboring nodes (e.g., the divisor):

*CurrentNode (E), Right (Y,E) $\Rightarrow$ CurrentNode (Y);

– unparse as above

4. Reevaluate the dividend and supply a default value for the divisor. E.g.,

CurrentNode (E), Left (X,E) $\Rightarrow$ Eval (X);

CurrentNode (E), Right (Y,E) $\Rightarrow$ Value (1, Y);

5. Abort evaluation by clearing out all Eval, Value and Check tuples:

*Eval (E) $\Rightarrow$;

*Value (V,E) $\Rightarrow$;

*Check (V,E) $\Rightarrow$;

All these functions (and more) could be provided as commands in a programming environment. In the next section we will investigate a command interpreter that permits debugging actions such as these.

8. Command Interpreter

In this section we will describe a simple command interpreter in Ω rules. This command interpreter will permit the interactive evaluation and unparsing of (already entered) abstract programs, in addition to various debugging and error recovery activities.

We assume the existence of a relation called Command that contains the last string or keystroke (we don't care which) typed on the keyboard. We will use boldfaced identifiers such as **evaluate** to represent commands; these identifiers could be bound to strings, key codes, menus coordinates. etc.

First we consider the **evaluate** command: its intended effect is to request evaluation of the current expression, which might be the entire program or some subexpression of it. This command is implemented by the following two rules:

*Command (**evaluate**), CurrentNode(E) $\Rightarrow$ Eval (E), Pendant (E).

*Pendant (E), *Value (V,E) $\Rightarrow$ Display {string$\leftarrow$ int [V]}.

The first rule detects the **evaluate** command and requests evaluation of the expression. That an evaluation is in progress is recorded in the Pendant relation. When a value arrives at the pendant node, it is displayed by the second rule.

The auxiliary relation Pendant can be eliminated by using Ω 's sequential mechanism:

*Command (**evaluate**), CurrentNode (E)
$\Rightarrow$ { Eval (E);
    *Value (V,E) $\Rightarrow$ Display {string$\leftarrow$ int [V]} }.

The commands in the curly braces are evaluated in order. Thus the tuple is asserted to Eval before the second rule waits for a tuple in Value.

The **val** command is used to explicitly attach a value to a node. This might be used during error recovery to allow evaluation to proceed in the face of errors. The command makes use of an additional relation Argument which holds a string value typed in from the keyboard. We can imagine this working as follows: The user types '254' and strikes the **val** key. This causes the string "254" to be put in

the Argument relation and the key **val** to be put in the Command relation. The rule for processing the **val** command is:

*Command(**val**), *Argument(V), CurrentNode(E)

$\Rightarrow$ Value (int$\leftarrow$string [V],E).

The **show** command requests the current expression to be unparsed and displayed. It is implemented by:

*Command (**show**), CurrentNode (E)

$\Rightarrow$ { Unparse (E);

*Image (S,E) $\Rightarrow$ Display {S}}.

It is also useful to have commands for moving within the abstract program structure. The **in** command "zooms in" by focusing on the left-argument of the current expression:

*Command (**in**), *CurrentNode (E), Left (X,E)

$\Rightarrow$ CurrentNode (X).

Thus the **in** command shifts the focus from the current node to the left-argument of the current node.

The **next** command shifts the focus from the left argument of an application to its right argument:

*Command (**next**), *CurrentNode (X), Left (X,E), Right (Y,E)

$\Rightarrow$ CurrentNode (Y).

In the pseudonatural notation this is expressed:

If given **next** is the command, node_1 is the current_node, node_1 is the left_argument of an appl_node, and node_2 is the right_argument of the appl_node then node_2 is the current_node.

Analogous commands are **prev**, which shifts from the right argument to the left argument, and **out** which "zooms out" from either the left of right argument to the entire application.

For debugging it is useful to be able to abort a suspended evaluation. This is accomplished by clearing out the Eval, Check and Value relations:

Command (**abort**), *Eval (E) $\Rightarrow$

else Command (**abort**), *Check (V,E) $\Rightarrow$

else Command (**abort**), *Value (V,E) $\Rightarrow$

else *Command (**abort**) $\Rightarrow$ Display {"aborted."}.

Notice (by the absence of an '*') that the first three rules leave **abort** in command; hence they continue firing as long as there are tuples in Eval, Check or Value. When there are no more such tuples, the last rule cancels the **abort** command.

It could be argued that these rules would be more readable if the reassertion of **abort** were made explicit, e.g.,

*Command (**abort**), *Eval (E) $\Rightarrow$ Command (**abort**)

else *Command (**abort**), *Check (V,E) $\Rightarrow$ Command (**abort**)

else *Command (**abort**), *Value (V,E) $\Rightarrow$ Command (**abort**)

else *Command (**abort**) $\Rightarrow$ Display {"aborted."}.

This is an unresolved stylistic issue.

To illustrate the operation of the command interpreter, we present an example session, showing the keys typed and the responses of the system. Assume the program '$((3\div0)+1)$' is already created and the current node is the root of the tree. The transcript follows:

evaluate

zero divide error

show

$(3\div0)$

in

evaluate

next

show

0

1 **val**

4

Notice how the **val** command triggers completion of evaluation of the program.

9. Syntax Directed Editing

9.1 Incomplete Programs

In this section we develop a syntax directed editor for this simple language. Since for editing we need to be able to represent incomplete programs we will add a new node type, 'Undef', representing a part of the program that either has not yet been entered or has been deleted:

$$expr = Con \cup Appl \cup Undef$$

Next we must modify Eval and Unparse to deal with Undef nodes:

*Eval(E), Undef(E), *CurrentNode($-$) $\Longrightarrow$ Display{"Incomplete"}, CurrentNode(E).

*Unparse(E), Undef(E) $\Longrightarrow$ Image ("< expr> ", E).

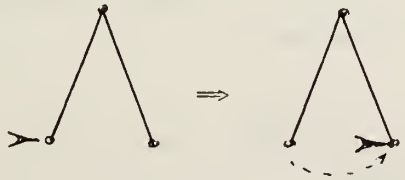
In other words, evaluating an incomplete program will lead to a diagnostic message and a suspended evaluation. Unparsing an incomplete program will show '< expr> ' in place of the missing subexpression.

9.2 Editor Commands

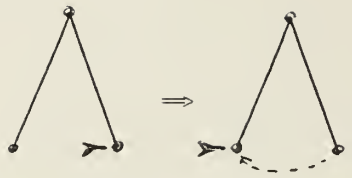
What commands do we want? We will want **in** to "zoom in" on a subexpression:

We will want **out** to "zoom out" from a subexpression:

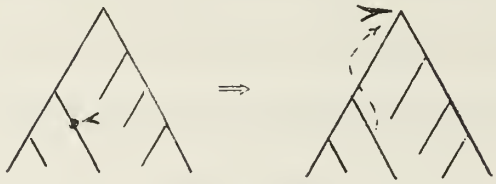
We will want **next** to shift to the next (to the right) subexpression:



We will want **prev** to shift to the previous (to the left) subexpression:

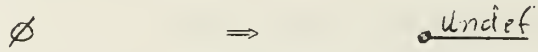


It will be convenient to have **root** to shift to the root of the program tree:

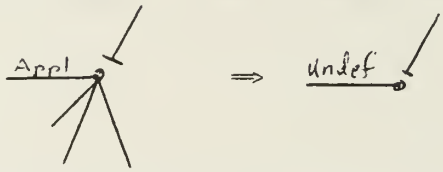


Furthermore, we will want all of these movement commands to show us the new current expression by unparsing it.

We will also need a **begin** command to initialize the editing session with an empty tree:

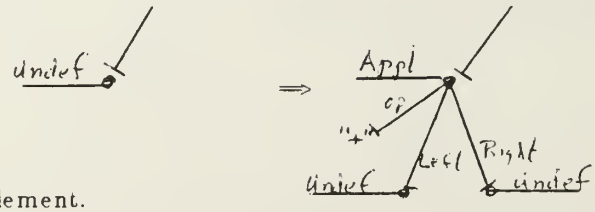


We will want to be able to **delete** a node:



And we will want to be able to insert a literal value, by '*n* #':

Finally we will need the operator commands, +, −, ×, ÷, for creating application nodes:



These are all simple to implement.

The predicate form for the **in** command rule is:

*Command(**in**), *CurrentNode(E), Left(X,E)

⟹ CurrentNode(X), Command(**show**).

This is the same as the version discussed in the previous section, except that we have automatically

issued a **show** command.

In the pseudonatural notation the **in** rule is:

If given "in" is the command, given an expression is the current_node, and a node is the left_argument of the expression

then the node is the current_node, and "show" is the command.

The **out** command is analogous, except that there are two rules to handle the two possible paths back to the parent:

*Command(**out**), *CurrentNode(X), Left(X,E)

$\Rightarrow$ CurrentNode(E), Command(**show**).

*Command(**out**), *CurrentNode(Y), Right(Y,E)

$\Rightarrow$ CurrentNode(E), Command(**show**).

The **next** and **prev** commands are similar.

The **delete** command is implemented by breaking the current node's connections with its descendants and changing its type to Undef. There are three cases depending on whether the node is a constant, application or already undefined node:

*Command(**delete**), CurrentNode(E), *Con(E), *Litval(V,E)

$\Rightarrow$ Undef(E), Command(**show**).

*Command(**delete**), CurrentNode(E),

*Appl(E), *Op(N,E), *Left(X,E), Right(Y,E)

$\Rightarrow$ Undef(E), Command(**show**).

*Command(**delete**), CurrentNode(E), Undef(E)

$\Rightarrow$ Display{"already deleted"}.

Note that subtrees are not disassembled; hence they could be reused (say by a move command).

There are two rules to implement the # command. The first one is to create a constant node with a given literal value:

*Command("#"), *Argument(V), **if** (integer(V)), CurrentNode(E). *Undef(E)

$\Rightarrow$ Con(E), Litval(V,E), Command(**show**).

The other rule handles the case where the current node is already defined; we require a node to be deleted before it can be replaced:

*Command("#"), *Argument(V), CurrentNode(E), $\neg$Undef(E)

$\Rightarrow$ Display{"defined node"}.

Next we consider that commands for creating application nodes. If the user types '+' then we must create an Appl node with two undefined daughters and a '+' for the operator:

*Command("+"), *CurrentNode(E), *Undef(E), *Avail(X, Y)

$\Rightarrow$ Appl(E), Op("+",E), Left(X,E), Right(Y,E),

Undef(X), Undef(Y), CurrentNode(X).

Here we have assumed that Avail contains an indefinite supply of unused objects; objects are allocated by a system procedure in the McArthur interpreter. Also notice that the focus is automatically shifted to the left argument of the new application.

It would be somewhat inconvenient to repeat the above rule for each of the four operators. Also we would need four rules for detecting already defined nodes. Fortunately we can use the applicative features of Ω to make one rule handle all four operators:

*Command(f), member [f, ["+", "$-$", "$\times$", "$\div$"]],

*CurrentNode(E), *Undef(E), *Avail(X, Y)

$\Rightarrow$ Appl(E), Op(f,E), Left(X,E), Right(Y,E), Undef(X), Undef(Y), CurrentNode(X).

Already defined nodes are handled by:

*Command(f), member [f, ["+", "$-$", "$\times$", "$\div$"]], CurrentNode(E), $\neg$Undef(E)

$\Rightarrow$ Display{"defined node"}.

The **begin** command is implemented by creating a tree containing a single undefined node:

*Command(**begin**), *CurrentNode($-$), *Avail(E) $\Rightarrow$ Root(E), Undef(E.

The **root** command is simple since the Root relation holds the root of the tree (set by the **begin** command):

*Command(**root**), *CurrentNode(−), Root(E)

$\Rightarrow$ CurrentNode(E), Command(**show**).

A typical session will illustrate operation of the syntax directed editor. Our goal will be to construct the program '$(3 \div 0)+1$' and change it to '$(3 \div 1)+1$'. We show commands on the left margin, and responses indented:

begin

< expr>

\+

÷

3 #

next

< expr>

0 #

out

(3÷0)

out

((3÷0) + < expr>)

in

(3÷0)

next

< expr>

1 #

root

((3÷0)+1)

evaluate

zero divide error

show

(3÷0)

in

3

next

0

delete

1 #

root

((3÷1)+1)

abort

aborted.

evaluate

4

This is, of course, a very simple system for a very simple language. But it illustrates the ideas of a programming environment. A version of this system that executes correctly under the McArthur interpreter [McArthur84] is shown in the appendices.

9.3 Permissions

We review the access to the various relations needed by the various tools:

- Editor — can read and update program structure relations (Con, Litval, etc.), CurrentNode, Root and evaluator and unparser relations (Eval, Check, Value, Unparse, Image).
- Evaluator — can only read program structure relations; can read and update evaluation relations (Eval, Check and Value); can update CurrentNode and Display; can read Meaning and Explanation.

- Unparser – can only read program structure relations; can read and update unparser relations (Unparse and Image); can read Template.

These rights can be enforced using the Ω capability mechanism; see [MacLennan83] or [MacLennan85] for a description.

10. Conclusions

We believe that this report has shown that major components of a programming environment, albeit in a rudimentary form, can be conveniently programmed in Ω. If this experience is typical, if a reasonable programming environment can be prototyped in a few hundred rules, then we believe that our ability to prototype software will have been much enhanced. Succeeding reports in this series will further investigate this hypothesis by expanding the capabilities of the prototype programming environment.

11. References

[MacLennan83] MacLennan, B. J., A View of Object-Oriented Programming, Naval Postgraduate School Computer Science Department Technical Report NPS52-83-001, February 1983.

[MacLennan84] MacLennan, B. J., The Four Forms of Ω: Alternate Syntactic Forms for an Object-Oriented Language, Naval Postgraduate School Computer Science Department Technical Report NPS52-84-026, December 1984.

[MacLennan85] MacLennan, B. J., A Simple Software Environment Based on Objects and Relations, *Proc. of ACM SIGPLAN 85 Conf. on Language Issues in Prog. Environments,* June 25-28, 1985, and Naval Postgraduate School Computer Science Department Technical Report NPS52-85-005, April 1985.

[McArthur84] McArthur, Heinz M., *Design and Implementation of an Object-Oriented, Production-Rule Interpreter,* MS Thesis, Naval Postgraduate School Computer Science Department, December 1984.

[Ufford85] Ufford, Robert P., *A Translation of an Extensible "Natural" Notation Language into an Object Oriented Language (Omega)* (tentative title). MS Thesis, Naval Postgraduate School Computer Science Department, June 1985.

APPENDIX A: Prototype Programming Environment

Predicate Notation for Ω

The following is a loadable input file for the prototype programming environment described in this report. It is accepted by the McArthur interpreter [McArthur84], which differs in a few details from the Ω described in this report (see [MacLennan84]). A transcript of a test execution of this environment is shown in Appendix C.

```
!              PI-1
!      Rules and associated definitions for
!      an arithmetic expression language.

!      Relations

! Program Structure Relations

define {root, "Appl", newrel{}};
define {root, "Op", newrel{}};
define {root, "Left", newrel{}};
define {root, "Right", newrel{}};
define {root, "Con", newrel{}};
define {root, "Litval", newrel{}};

! Evaluation Relations

define {root, "Eval", newrel{}};
define {root, "Check", newrel{}};
define {root, "Value", newrel{}};
define {root, "Meaning", newrel{}};
define {root, "Explanation", newrel{}};

! Unparser Relations
```

```
define {root, "Unparse", newrel{}};

define {root, "Image", newrel{}};

define {root, "Template", newrel{}};


! Command Interpreter Relations

define {root, "Command", newrel{}};

define {root, "Argument", newrel{}};

define {root, "Root", newrel{}};

define {root, "Undef", newrel{}};

define {root, "CurrentNode", newrel{}};

define {root, "EvalPending", newrel{}};

define {root, "ShowPending", newrel{}};

define {root, "CreateAppl", newrel{}};

define {root, "CreateRoot", newrel{}};

define {root, "Script", newrel{}};

define {root, "PendScript", newrel{}}.


!       Functions

fn Id [x]: x;

fn Sum [x,y]: x + y;

fn Dif [x,y]: x - y;

fn Product [x,y]: x * y;

fn Quotient [x,y]:

   if y = 0 -> ["error", 1]

   else x / y;

fn IsErrorcode [w]:
```

```
    if ~IsList[w] | w = Nil -> Nil
    else first[w] = "error";

fn upSum [x,y]: "(" + x + " + " + y + ")";
fn upDif [x,y]: "(" + x + " - " + y + ")";
fn upProd [x,y]: "(" + x + " x " + y + ")";
fn upQuot [x,y]: "(" + x + " / " + y + ")".

!     Built-in Tables

Meaning (Sum, "+ ");
Meaning (Dif, "-");
Meaning (Product, "x");
Meaning (Quotient, "/");
Meaning (Id, "lit");

Template (upSum, "+ ");
Template (upDif, "-");
Template (upProd, "x");
Template (upQuot, "/");
Template (int_str, "lit");

Explanation ("incomplete program", ["error", 0]);
Explanation ("division by zero", ["error", 1]).

!     the Rules

define{root, "PI1Rules",
<<
! Evaluator Rules

! Constant nodes
```

```
if *Eval(e), Con(e), Litval(v,e), Meaning(f, "lit")
-> Value(f[v], e);

! Appl nodes

if *Eval(e), Appl(e), Left(x,e), Right(y,e)
-> Eval(x), Eval(y);

if *Value(u,x), *Value(v,y), Appl(e), Op(n,e), Left(x,e), Right(y,e), Meaning(f, n)
-> Check(f[u,v], e);

! Error Checking

if *Check(w, e), ~IsErrorcode[w]
-> Value(w, e);

if *Check(w, e), IsErrorcode[w], Explanation(s, w), *CurrentNode(q)
-> displayn{s}, CurrentNode(e);

! Unparser

! Constant Nodes

if *Unparse(e), Con(e), Litval(v,e), Template(f, "lit")
-> Image(f[v], e);

! Identifier nodes

! Appl nodes

if *Unparse(e), Appl(e), Left(x,e), Right(y,e)
-> Unparse(x), Unparse(y);

if *Image(u,x), *Image(v,y), Appl(e), Op(n,e), Left(x,e), Right(y,e), Template(f, n)
-> Image(f[u,v], e);
```

```
! Command Interpreter Rules

! evaluate Command

if *Command("evaluate"), CurrentNode(E)
-> Eval(E), EvalPending(E);

if *Value(V,E), *EvalPending(E)
-> displayn {V};

! return Command

if *Command("val"), *Argument(V), CurrentNode(E)
-> Value(V,E);

! show Command

if *Command("show"), CurrentNode(E)
-> Unparse(E), ShowPending(E);

if *Image(S,E), *ShowPending(E)
-> displayn{S};

! abort Command

if Command("abort"), *Eval(E) -> ;

if Command("abort"), *Value(V,E) -> ;

if Command("abort"), *Check(V,E) -> ;

if *Command("abort"), ~Eval(E), ~Value(V,E) -> displayn{"aborted"};

! Handle incomplete program

if *Eval(E), Undef(E), *CurrentNode(Q)
```

```
-> displayn("Incomplete"), CurrentNode(E);

if *Unparse(E), Undef(E)

-> Image ("< expr> ", E);

! Syntax Directed Editing

! in Command

if *Command("in"), *CurrentNode(E), Left(X,E)

-> CurrentNode(X), Command("show");

if *Command("out"), *CurrentNode(X), Left(X,E)

-> CurrentNode(E), Command("show");

if *Command("out"), *CurrentNode(Y), Right(Y,E)

-> CurrentNode(E), Command("show");

! next Command

if *Command("next"), *CurrentNode(X), Left(X,E), Right(Y,E)

-> CurrentNode(Y), Command("show");

! prev Command

if *Command("prev"), *CurrentNode(Y), Right(Y,E), Left(X,E)

-> CurrentNode(X), Command("show");

! delete command

if *Command("delete"), CurrentNode(E), *Con(E), *Litval(V,E)

-> Undef(E), Command("show");

if *Command("delete"), CurrentNode(E), *Appl(E), *Op(N,E), *Left(X,E), Right(Y,E)

-> Undef(E), Command("show");
```

```
if *Command("delete"), CurrentNode(E), Undef(E)
-> displayn("already deleted");

! # Command

if *Command("#"), *Argument(V), IsInt[V], CurrentNode(E), *Undef(E)
-> Con(E), Litval(V,E), Command("show");

if *Command("#"), *Argument(V), CurrentNode(E), ~Undef(E)
-> displayn("defined node");


! +, -, x , / Commands

if *Command(op), member [op, ["+ ", "-", "x", "/"]], *CurrentNode(E), *Undef(E)
-> CreateAppl(op, E, newobj{}, newobj{});

if *CreateAppl(op,E,X,Y)
-> Appl(E), Op(op,E), Left(X,E), Right(Y,E), Undef(X), Undef(Y), CurrentNode(X);

if *Command(op), member [op, ["+ ", "-", "x", "/"]], CurrentNode(E), ~Undef(E)
-> displayn("defined node");

! begin Command

if *Command("begin"), *CurrentNode(Q)
->  CreateRoot(newobj{});

if *CreateRoot(E)
-> Root(E), Undef(E), CurrentNode(E);

! root Command

if *Command("root"), *CurrentNode(Q), Root(E)
-> CurrentNode(E), Command("show");
```

```
! Test Driver

if *Script(Nil) -> displayn{"Script completed"}

else if *Script(L), (first[L] = "#" | first[L] = "val")
-> { displayn {" ... " + first [rest [L]] + first [L]};
    Command(first[L]), Argument(first[rest[L]]), PendScript(rest[rest[L]]) }

else if *Script(L)
-> { displayn {" ... " + first [L]};
    Command(first[L]), PendScript(rest[L]) };

if *PendScript(L), ~Command(Q) -> Script(L)

>> }.

define {root, "testscript",
 ["begin", "+ ", "/", "#", 3, "next", "#", 0, "out", "out",
  "in", "next", "#", 1, "root", "evaluate", "show", "in",
  "next", "delete", "#", 1, "root", "abort", "evaluate"] }.

!      activate the rules

act{ PI1Rules }.
CurrentNode(Nil).
displayn{"PI-1 System loaded"}.
```

APPENDIX B: Prototype Programming Environment

Pseudonatural Notation for Ω

This appendix displays the prototype programming environment of Appendix A in the pseudonatural notation designed by Robert Ufford [Ufford85]. Ufford also performed this translation of the Appendix A program into the pseudonatural notation.

```
!
                PI-1
    Rules and associated definitions for
    an arithmetic expression language.
!

!   Relations   !

!   Program structure relations   !

"Application" (procedure) is defined as a relation.
"Operator" (procedure) is defined as a relation.
"Left_argument" (procedure) is defined as a relation.
"Right_argument" (procedure) is defined as a relation.
"Constant" (procedure) is defined as a relation.
"Literal_value" (procedure) is defined as a relation.

!   Evaluation relations   !

"Evaluated" (procedure) is defined as a relation.
"Checked" (procedure) is defined as a relation.
"Value" (procedure) is defined as a relation.
"Meaning" (procedure) is defined as a relation.
"Explanation" (procedure) is defined as a relation.
```

```
! Unparser relations !

"Unparsed" (procedure) is defined as a relation.

"Image" (procedure) is defined as a relation.

"Template" (procedure) is defined as a relation.

! Command interpreter relations !

"Command" (procedure) is defined as a relation.

"Argument" (procedure) is defined as a relation.

"Root_node" (procedure) is defined as a relation.

"Undefined" (procedure) is defined as a relation.

"Current_node" (procedure) is defined as a relation.

"Pending_evaluation" (procedure) is defined as a relation.

"Shown" (procedure) is defined as a relation.

"New_application" (procedure) is defined as a relation.

"New_root" (procedure) is defined as a relation.

"Script" (procedure) is defined as a relation.

"Pending_script" (procedure) is defined as a relation.


! Functions !

function identity [x]: x.

function sum [x,y]: x + y.

function difference [x,y]: x - y.

function product [x,y]: x * y.

function quotient [x,y]:

    if y = 0 then the_list of the "error_code" and 1

    else x / y.


function error_code [W]:
```

```
    if W (predicate) is not a_list  W = Nil then Nil
    else the first (function) of W = "error_code".

function sum_template [x,y]: "(" + x + " + " + y + ")".
function difference_template [x,y]: "(" + x + " - " + y + ")".
function product_template [x,y]: "(" + x + " x " + y + ")".
function quotient_template [x,y]: "(" + x + " / " + y + ")".

!  Built-in tables  !

Sum is the meaning of "+ ".
Difference is the meaning of "-".
Product is the meaning of "x".
Quotient is the meaning of "/".
Identity is the meaning of "lit".

Sum_template is a template for "+ ".
Difference_template is a template for "-".
Product_template is a template for "x".
Quotient_template is a template for "/".
String_notation is a template for "lit".

"Incomplete program" is an explanation for the_list of error_code and 0.
"Division by zero" is an explanation for the_list of error_code and 1.

!  Noise words  !

"Must" (procedure) is defined as a noise_verb.
"Be" (procedure) is defined as a noise_verb.
"Being" (procedure) is defined as a noise_verb.
"Established" (procedure) is defined as a noise_verb.
"Will" (procedure) is defined as a noise_verb.
```

"Another" (procedure) is defined as a noise_prep.

! The rules !

"Pl1_rules" (procedure) are defined as

Rules

! Evaluator rules !

! Constant nodes !

If given an expression is being evaluated,

the expression is a constant,

a number is the literal_value of the expression, and

a lit_function is the meaning of "lit"

then the lit_function (function) of the number is the value of the expression;

! Application nodes !

If given an expression is being evaluated,

the expression is an application,

node1 is the left_argument of the expression, and

node2 is the right_argument of the expression

then node1 must be evaluated, and

node2 must be evaluated;

If given value1 is the value of node1,

given value2 is the value of node2,

the expression is an application,

a string is the operator of the expression,

node1 is the left_argument of the expression,

node2 is the right_argument of the expression, and

an operator_function is the meaning of the string

then the operator_function (function) of value1 and value2 must be checked

for the expression;

! Error checking !

If given an alleged_value is being checked for an expression, and

the alleged_value (predicate) is not an error_code

then the alleged_value is the value of the expression;

If given an alleged_value is being checked for an expression,

the alleged_value (predicate) is the error_code,

a string is an explanation for the alleged_value , and

given any_node is the current_node

then the string (procedure) is displayed_with_return and

the expression is the current_node;

! Unparser !

! Constant Nodes !

If given an expression is being unparsed,

the expression is a constant,

value1 is the literal_value of the expression, and

a lit_function is a template for "lit"

then the lit_function (function) of value1 is the image of the expression;

! Identifier nodes !

! Application nodes !

If given an expression is being unparsed,

the expression is an application,

node1 is the left_argument of the expression, and

node2 is the right_argument of the expression

then node1 must be unparsed and

node2 must be unparsed;

If given image1 is the image of node1,

given image2 is the image of node2,

the expression is an application,

a string is the operator of the expression,

node1 is the left_argument of the expression,

node2 is the right_argument of the expression, and

an operator_function is a template for the string

then the operator_function (function) of image1 and image2

is the image of the expression;

! Command interpreter rules !

! Evaluate command !

If given "evaluate" is the command, and

an expression is the current_node

then the expression must be evaluated, and

the expression is pending_evaluation;

If given value1 is the value of an expression, and

the expression is pending_evaluation

then value1 (procedure) is displayed_with_return;

! Return command !

If given "val" is the command,

given value1 is the argument, and

an expression is the current_node

then value1 is the value of the expression;

! Show command !

If given "show" is the command, and

an expression is the current_node

then the expression must be unparsed, and

the expression will be shown;

If given a string is the image of an expression, and

given the expression must be shown

then the string (procedure) is displayed_with_return;

! Abort command !

If "abort" is the command, and

given an expression is being evaluated

then !do nothing! .

If "abort" is the command, and

given a_value is the value of an expression

then !do nothing! .

If "abort" is the command, and

given a_value is being checked for an expression

then !do nothing! .

If given "abort" is the command,

an expression is not being evaluated, and

a_value is not the value of the expression

then "aborted" (procedure) is displayed_with_return;

```
!  Handle incomplete program  !

If given an expression is being evaluated,
   the expression is undefined, and
   given any_node is the current_node
then "Incomplete" (procedure) is displayed_with_return, and
   the expression is the current_node;

If given an expression is being unparsed, and
   the expression is undefined
then "< expr> " is the image of the expression;

!  Syntax Directed Editing  !

!  in Command  !

If given "in" is the command,
   given an expression is the current_node, and
   node1 is the left_argument of the expression
then node1 is the current_node, and
   "show" is the command;

If given "out" is the command,
   given node1 is the current_node, and
   node1 is the left_argument of an expression
then the expression is the current_node, and
   "show" is the command;

If given "out" is the command,
   given node2 is the current_node, and
   node2 is the right_argument of an expression
then the expression is the current_node, and
```

"show" is the command;

! next Command !

If given "next" is the command,
given node1 is the current_node,
node1 is the left_argument of an expression, and
node2 is the right_argument of the expression
then node2 is the current_node, and
"show" is the command;

! prev Command !

If given "prev" is the command,
given node2 is the current_node,
node2 is the right_argument of an expression, and
node1 is the left_argument of the expression
then node1 is the current_node, and
"show" is the command;

! delete command !

If given "delete" is the command,
an expression is the current_node,
given the expression is a constant, and
given a_value is the literal_value of the expression
then the expression is undefined, and
"show" is the command;

If given "delete" is the command,
an expression is the current_node,
given the expression is an application,

given a string is the operator of the expression,

given node1 is the left_argument of the expression, and

node2 is the right_argument of the expression

then the expression is undefined, and

"show" is the command;

If give "delete" is the command,

an expression is the current_node, and

the expression is undefined

then "already deleted" (procedure) is displayed_with_return;

! # Command !

If given "#" is the command,

given value1 is the argument,

value1 (predicate) is an_integer,

an expression is the current_node, and

given the expression is undefined

then the expression is a constant,

value1 is the literal_value of the expression, and

"show" is the command;

If given "#" is the command,

given value1 is the argument,

an expression is the current_node, and

the expression is not undefined

then "defined node" (procedure) is displayed_with_return;

! +, -, x, / Commands

If given a string is the command,

the string is a member of the_list of "+ ", "-", "x", and "/",

given an expression is the current_node, and

given the expression is undefined

then the expression is established as a new_application with a string

and an object and another object;

If given an expression is a new_application with a string and node1

and node2

then the expression is an application,

the string is the operator of the expression,

node1 is the left_argument of the expression,

node2 is the right_argument of the expression,

node1 is undefined,

node2 is undefined, and

node1 is the current_node;

If given a string is the command,

the string is a member of the_list of "+ ", "-", "x", and "/",

an expression is the current_node, and

the expression is not undefined

then "defined node" (procedure) is displayed_with_return;

! begin Command !

If given "begin" is the command, and

given any_node is the current_node

then an object is established as a new_root;

If given an expression is a new_root

then the expression is a root_node,

the expression is undefined, and

```
  the expression is the current_node;

!  root Command  !

If given "root" is the command,
  given any_node is the current_node, and
  an expression is the root_node
then the expression is the current_node, and
  "show" is the command;

!  Test driver  !

If given Nil is the script
then "Script completed" (procedure) is displayed_with_return

Else if given a list is the script, and
  (the first (function) of the list = "#"|
                the first (function) of the list = "val")
then
 begin
  " ... " (procedure) is displayed;
  the first (function) of the rest (function) of the list
    (procedure) is displayed;
  the first (function) of the list (procedure) is
    displayed_with_return;
  the first (function) of the list is the command,
  the first (function) of the rest (function) of the list is the argument, and
  the rest (function) of the rest (function) of the list is the pending_script
 end_block;

Else if given a list is the script
```

```
then
  begin
    " ... " (procedure) is displayed;
    the first (function) of the list (procedure) is
      displayed_with_return;
    the first (function) of the list is the command, and
    the rest (function) of the list is the pending_script
  end_block;

If given a list is the pending_script, and
   something is not the command
then the list is the script;

end_rules.

!   activate the rules   !

The PI1_rules (procedure) are activated.
Nil is the current_node.
"PI-1 System loaded" (procedure) is displayed_with_return.
```

APPENDIX C: Transcript of Ω Session

The following is a transcript of an Ω session illustrating the operation of the prototype programming environment shown in Appendix A. The assertion 'Script (testscript)' causes the commands in testscript to be executed in order. Each command is printed on a separate line, followed by whatever output is generated by the programming environment. This transcript was produced by the McArthur interpreter [McArthur84].

```
OMEGA-1  11/30/84

Use Cntl-D or exit{} to quit.

For help, enter help{"?"}.

To report a bug, enter Bugs{}.
PI-1 System loaded
> Script (testscript).
[begin, +, /, #, 3, next, #, 0, out, out, in, next, #, 1, root, evaluate, show, in, in, next,
 delete, #, 1, root, abort, evaluate]
 ... begin
 ... +
 ... /
 ... 3#
3
 ... next
<expr>
 ... 0#
0
 ... out
(3 / 0)
 ... out
```

((3 / 0) + < expr>)

... in

(3 / 0)

... next

< expr>

... 1#

1

... root

((3 / 0) + 1)

... evaluate

division by zero

... show

(3 / 0)

... in

3

... next

0

... delete

< expr>

... 1#

1

... root

((3 / 1) + 1)

... abort

aborted

... evaluate

4

Script completed

```
> exit{}.

Goodbye.
```

INITIAL DISTRIBUTION LIST

Defense Technical Information Center Cameron Station Alexandria, VA 22314	2
Dudley Knox Library Code 0142 Naval Postgraduate School Monterey, CA 93943	2
Office of Research Administration Code 012A Naval Postgraduate School Monterey, CA 93943	1
Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, CA 93943	40
Associate Professor Bruce J. MacLennan Code 52ML Department of Computer Science Naval Postgraduate School Monterey, CA 93943	12
Dr. Robert Grafton Code 433 Office of Naval Research 800 N. Quincy Arlington, VA 22217-5000	1
Dr. David Mizell Office of Naval Research 1030 East Green Street Pasadena, CA 91106	1
Professor Jack M. Wozencraft, 62Wz Department of Electrical and Comp. Engr. Naval Postgraduate School Monterey, CA 93943	1
Professor Rudolf Bayer Institüt für Informatik Technische Universität Postfach 202420 D-8000 Munchen 2 West Germany	1
Dr. Robert M. Balzer USC Information Sciences Inst. 4676 Admiralty Way Suite 10001 Marina del Rey, CA 90291	1

Mr. Ronald E. Joy
Honeywell, Inc.
Computer Sciences Center
10701 Lyndale Avenue South
Bloomington, MI 55402 1

Mr. Ron Laborde
INMOS
Whitefriars
Lewins Mead
Bristol
Great Britain 1

Mr. Lynwood Sutton
Code 424, Building 600
Naval Ocean Systems Center
San Diego, CA 92152 1

Mr. Jeffrey Dean
Advanced Information and Decision Systems
201 San Antonio Circle, Suite 286
Mountain View, CA 94040 1

Mr. Jack Fried
Mail Station D01/31T
Grumman Aerospace Corporation
Bethpage, NY 11714 1

Mr. Dennis Hall
New York Videotext
104 Fifth Avenue, Second Floor
New York, NY 10011 1

Professor S. Ceri
Laboratorio di Calcolatori
Departimento di Elettronica
Politecnico di Milano
20133 - Milano
Italy 1

Mr. A. Dain Samples
Computer Science Division - EECS
University of California at Berkeley
Berkeley, CA 94720 1